1987

A Software Metrics Survey

S. D. Conte

Herbert E. Dunsmore
Purdue University, dunsmore@cs.purdue.edu

V. Y. Shen

W. M. Zage

Report Number:
87-720

A SOFTWARE METRICS SURVEY

S. D. Conte
H. E. Dunsmore
V. Y. Shen
W. M. Zage

CSD-TR-720
February 1987

A SOFTWARE METRICS SURVEY

S. D. Conte, H. E. Dunsmore, V. Y. Shen, and W. M. Zage
Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

CSD TR-720
February 16, 1987

This paper was adapted from the book *Software Engineering Metrics and Models*, Benjamin/Cummings Publishing Company, Menlo Park, CA (1986), by S.D. Conte, H.E. Dunsmore, and V.Y. Shen.

1. INTRODUCTION
2. SIZE METRICS
   - 2.1 Lines of Code
   - 2.2 Token Count
   - 2.3 Function Count
   - 2.4 Equivalent Size Measure
3. DATA STRUCTURE METRICS
   - 3.1 The Amount of Data
   - 3.2 The Usage of Data within a Module
4. LOGIC STRUCTURE METRICS
   - 4.1 Decision Count
   - 4.2 Minimum Number of Paths and Reachability Metrics
   - 4.3 Nesting Levels
   - 4.4 Transfer Usage
5. EFFORT AND DEFECT METRICS
   - 5.1 Measuring Effort
   - 5.2 Measuring Defects
6. SOME APPLICATIONS OF SOFTWARE METRICS
   - 6.1 Early Software Size Estimation
   - 6.2 Guiding the Testing Process
   - 6.3 Using Many Metrics to Determine Software Complexity
7. SUMMARY AND CONCLUSIONS

A SOFTWARE METRICS SURVEY

1. INTRODUCTION

Donald Knuth has established an ideal for software developers with the following statement [KNUT73]: "Some programs are elegant, some are exquisite, some are sparkling. My claim is that it is possible to write *grand* programs, *noble* programs, truly *magnificent* ones!" The measurement of the eloquence and grandeur of a program will most likely always remain a subjective judgment. However, there exist objective metrics that could help one estimate such practical characteristics as the size and cost of software development. Furthermore, there are metrics which could potentially lead to a better understanding of the software development process.

The full realization of the potential of computers depends critically on our ability to produce reliable software at a reasonable cost. Despite the claims of our ability to produce "elegant", "exquisite", or "truly magnificent" programs by software experts, there is great national concern that software technology lags so far behind hardware technology that this potential will never be fully realized. Numerous reports document the observation that while the cost of hardware to perform a given function continues to decrease dramatically, the cost of software required for that function has continued to increase. This cost is growing both in absolute terms and as a percentage of the total data processing budget.

The techniques of software engineering have been introduced in an attempt to counteract these software cost trends. Engineering as a discipline has always relied heavily on metrics and models to help it arrive at quantitative assessments of the cost and quality of products. On the other hand, computer software, until recently, has been perceived to be impervious to these engineering approaches, with software development viewed more as an art than a science. Cost overruns, project delays, and poor reliability which have characterized software (especially large-scale software) have been accepted in the past as the norm.

There is now a growing recognition that managers must focus on the entire life cycle process from requirements through maintenance if we are to control the spiraling cost of software. The proper use of software metrics and models is essential in the successful management of software development and maintenance. Software metrics are used to characterize quantitatively the essential features of software. After a number of useful metrics is identified, it is then possible to measure software in an algorithmic and objective fashion, so that the values of the selected metrics are consistent among different software products and are independent of the measurer. In order to control the software development and maintenance processes, it is possible to estimate some interesting metrics such as effort (i.e., cost) and defects based on other metrics that are available. Appropriate management decisions can be made to influence these factors so that management goals can be met. In other words, the proper use of software metrics and models has the potential of allowing us to recognize the development of "truly magnificent" software objectively, and to estimate the added cost (if any) of producing such software accurately. It is fair to say, however, that the technology is a long way from realizing these potential benefits. Although significant progress in the development of metrics and models has been achieved in recent years, a great deal of research is still required to refine and validate existing models and to develop new and better ones.

There are many things one can measure about computer software: the size in lines of code, the cost of development in dollars, the time for development in work-days, the memory requirement in bytes and even the number of customer complaints received after delivery. Sections 2, 3 and 4 of this paper discuss the metrics related to a program's size, data structure, and logic structure. These metrics are referred to as "product" metrics since they can be derived from analyzing the software itself using an automatic tool. The tool is often a computer program called a *software analyzer* that accepts the program as input and provides counts of various metrics as output.

There are other measurements related to a software product which depend mostly on the development environment. These are called "process" metrics since they are based on the development process and require no observation of the program itself. An example is the measure of time taken by a programmer to design, code, and test a program. This measurement depends on (among many other things) the difficulty of the assignment, the ability of the programmer, the methodology used, and the availability of the computer during the development process. Other important process metrics include the effort (cost) of software development and the method by which defects are observed. Process metrics are discussed in Section 5 of this paper.

Space limitations prevent us from being completely exhaustive in our software metrics survey that follows. The absence of a metric should not be interpreted as a negative evaluation of its quality. However, more emphasis is given to those metrics that are employed most often and that have been found to be the most useful by software researchers.

2. SIZE METRICS

The size of a computer program used to be determined easily by the number of punched cards it took to contain the program. This metric is roughly equivalent to the *lines of code* metric that survives to this day. The size of a program is an important measure for primarily these reasons:

- It is easy to compute after the program is completed.
- It may possibly be estimated before completion.
- It is the most important factor for many software cost models.
- Productivity is normally based on a size measure.

Although the lines of code measure of size is very popular, it may not be satisfactory for modern programming languages since some lines (or groups of lines) in a program are more

difficult to produce than other lines in the same program. (For example, in the Fortran subroutine displayed in Figure 1, line 7 is probably easier to construct than line 9.) Such concern about consistency leads to two opposite approaches:

(1) Refine the size measure by counting the basic tokens on all lines and ignoring the lines themselves. (For example, line 8 contains the seven tokens DO, 220, J, =, 1, the comma, and IM1.)

(2) Generalize the measure by grouping lines that support well-defined functions. (For example, lines 9-12 assure that X(I) and X(J) are in the proper order.)

The following subsections discuss measures based on lines of code, count of tokens, and count of functions. All of these have been used for cost estimation, software comparison and productivity studies.

2.1 Lines of Code

The unit of a program's size in lines of code is (obviously!) LOC. We denote this by the symbol S_s, for small programs. For larger programs, it is often appropriate to measure the size in thousands of lines of code (KLOC), which we represent by S. Although this may seem to be a simple metric that can be counted objectively in a straightforward way, there is no general agreement about what constitutes a line of code. For example, in the Fortran program shown in Figure 1, if S_s is simply a count of the number of lines, then this program contains 16 LOC. But most measurement specialists agree that the line of code measure should not include comments or blank lines since these are really internal documentation and their presence or absence does not affect the functions of the program. More importantly, comments and blank lines are not as difficult to construct as program lines. The inclusion of comments and blank lines in the count may encourage programmers to introduce many such lines in project development in order to create the illusion of high productivity, which is normally measured in LOC/PM (lines-of-

code/person-month). When comments and blank lines are ignored, the program in Figure 1 contains 14 lines. Furthermore, if the main interest is the size of the program that supports a certain function, it may be reasonable to include only executable statements in the S_s count. The only executable statements in Figure 1 are in lines 5-15 leading to an S_s count of 11. Thus the differences in the counts for Figure 1 are 16 to 14 to 11. One can easily see the potential for major discrepancies for large programs with many comments or programs written in languages that allow (or even require) a large number of descriptive but non-executable statements. Furthermore, there is another problem in counting the size of a program written in a language that permits free-format coding. These languages often allow compounding with two or more statements on one line or a single statement extended over two or more lines. A well-accepted solution to this inconsistency is a standard definition for lines of code:

A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements.

By this definition, the program in Figure 1 has 14 LOC.

2.2 Token Count

At the beginning of Section 2 we discussed a major problem with the lines of code measure: it is not consistent because some lines are more difficult to code than others. One solution to this problem is to give more weight to lines that have more "tokens" – the basic syntactic units distinguishable by a compiler. Such a scheme was used by Halstead in his family of metrics commonly called *Software Science* [HALS77]. A computer program is considered in Software Science to be a collection of tokens that can be classified as either *operators* or *operands*. All Software Science measures are functions of the counts of these tokens. The basic metrics are defined as:

η_1 = number of unique operators (1)

η_2 = number of unique operands (2)

N_1 = total occurrences of operators (3)

N_2 = total occurrences of operands (4)

Generally, any symbol or keyword in a program that specifies an action is considered an operator, while a symbol used to represent data is considered an operand. Most punctuation marks are also categorized as operators. Variables, constants, and even labels are operands. Operators consist of arithmetic symbols (such as +, -, and /), command names (such as WHILE, FOR, and READ), special symbols (such as :=, braces, and parentheses), and even function names (such as SORT in Figure 1).

The size of a program in terms of the total number of tokens used (called "length" by Halstead) is

$$N = N_1 + N_2 \tag{5}$$

Table 1 shows a sample operator and operand analysis of the program in Figure 1. Just as there are variations in counting lines of code, there are variations in classifying operators and operands. The original rules established by Halstead excluded the counting of declaration statements and input/output statements. Statement labels were not counted as operands, but rather were considered a part of direct transfers (for example, a line such as GOTO 200 was considered a unique operator). Since there is currently no general agreement among researchers on what is the most "meaningful" way to classify and count these tokens [SHEN83], counts may vary among software analyzers. We have chosen to include the declarations and statement labels in Table 1 for convenience.

In Table 1 note that $N_1 = 51$ and $N_2 = 42$ and the length metric $N = N_1 + N_2 = 93$. The Software Science N may be converted to an estimate of S_s via the relationship $S_s = N/c$, where the constant c is language-dependent. For Fortran, it is thought to be about 7. Notice that

93/7 = 13, which is close to the actual value of S_s (14) in Figure 1. A study of over one thousand commercial assembly language and PL/S modules found that S_s and N were indeed linearly related, and appeared equally valid as relative measures of program size [CHRI81].

2.3 Function Count

Many researchers have found units larger than lines of code useful measures for characterizing the size of programs, especially for large programming products. Some have tried to use the *module* as a unit - usually defined as a segment of code that is independently compilable. For large programs, it may be easier to predict the number of modules than the number of lines of code. Most of us divide a program into modules based upon various considerations. One is that a module should be the subprogram for one algorithm (such as input the data, find the maximum value in the table, insert a new item in the inventory). Another consideration is that a module should consist of all operations on one data structure (such as all push, pop, and read operations for a stack, or all make, check, and cancel reservation operations for an airline flight). With these ways of constructing modules, there is a large variation in module size. This phenomenon has been observed in a study of a large number of commercial products, where the sizes of modules ranged from less than 10 to nearly 10,000 lines of code [SMIT80].

A function in a program is defined as a collection of executable statements that performs a certain task, together with declarations of the formal parameters and local variables manipulated by those statements. A function is an abstraction of *part* of the tasks that the program is to perform. This idea is based on the observation that a programmer may think in terms of building a program from functions, rather than from statements or even modules. For example, if you show Figure 1 to an experienced programmer and ask what it does, the experienced programmer might report that this is an "interchange sort." Thus, to an experienced programmer, this subroutine does not contain several lines of code, but a single function. In fact, most modules can be divided into one or more functions by an experienced programmer.

Some authors advocate that module sizes be limited to 50-200 LOC, arguing that such sizes increase understandability and minimize errors [BAKE72, BELL74]. The overhead involved, both mental and syntactic, discourages the definition of functions that are too small. It is probably easier to put together three 40-line functions to make a program, than to put together twelve 10-line functions or 120 one-line functions. Thus, the variation of size in LOC for functions may not be as great as that for modules. Indirect support is found in a study of student projects, which shows that programmers use a similar number of functions to solve a given problem, but use a different number of modules [BASI79].

In [ALBR83] the authors use a count of function points in effort estimation. The use of metrics as predictive tools is discussed further in Section 6 of this paper.

2.4 Equivalent Size Measures

It is important to realize that programmers do not always develop *new* software. In fact, a good deal of modern software development involves modifying existing code. This is done in order to produce a product that is new in the sense that it does things that the previous software did not, but not totally new in the sense that it involves software borrowed from a previous version or from a similar program. The current situation in industry is such that 50-95% of what programmers do is to modify existing programs. In these cases, they add new code while re-using old code. It has been reported that, at IBM's Santa Teresa Laboratory, 77% of all program code is written in order to add new features to existing products [PAUL83].

Thus for many programs size has two components: S_n for newly written code and S_u for code adapted from existing software (or re-used code). The components of size may be expressed in any of the previously discussed metrics (line of code, token count, or function count). In any case what is desired is an equivalent size measure S_e, which is a function of S_n and S_u. This means that the effort required to develop the software with S_n and S_u (new and re-used code) is "equivalent" to the effort of developing a product with S_e "from scratch" (all

new code and no re-used code). Models to determine S_e range from linear functions in [BOEH81] and [BAIL81] (Bailey suggests using the formula $S_e = S_n + 0.2S_u$) to non-linear ones such as that in [THEB83]. At this time there is no general consensus as to how to compute S_e.

3. DATA STRUCTURE METRICS

A set of metrics that capture the amount of data input to, processed in, and output from software are called the *data structure metrics*. The importance of such metrics can be seen intuitively in the following example: assuming that a problem can be solved in two ways, resulting in two programs A and B. A has 25 input parameters, 35 internal data items, and 10 output parameters. B has 5 input parameters, 12 internal data items, and 4 output parameters. We can surmise that A is probably more complicated, took more time to program, and has a greater probability of errors than B.

This section presents several data structure metrics. Some concentrate on variables (and even constants) within each module and ignore the input/output dependencies. Others concern themselves primarily with the input/output situation. Since there is no general agreement on how the lines of code measure is to be counted, one should not be surprised that there are various methods for measuring data structures as well.

3.1 The Amount of Data

Most compilers and assemblers have an option to generate a cross-reference list, indicating the line where a certain variable is declared and the line or lines where it is referenced. For example, a cross-reference listing for the sorting subroutine that appears in Figure 1 is given in Figure 2.

One method for determining the amount of data is to count the number of entries in the cross-reference list. Such a count of variables will be referred to as *VAR*. For the subroutine appearing in Figure 1, $VAR = 6$. The count of variables *VAR* depends on the following

definition:

A variable is a string of alphanumeric characters that is defined by a programmer and that is used to represent some value during either compilation or execution.

Although a simple way to obtain *VAR* is from a cross-reference list, it can also be generated using a software analyzer that counts the individual tokens, as described in Section 2.2. It may be appropriate to exclude the variables that are defined but not referenced within the software module (such as the subroutine name SORT).

A similar measure of data is the *unique operands* metric of Software Science defined as η_2 in Equation (2). In addition to the unique variables, it also includes constants and labels. That is,

$$\eta_2 = VAR + \text{unique constants} + \text{labels}.$$

The sample SORT program in Figure 1, which is analyzed in Table 1, has 6 variables (X, N, I, J, SAVE, IM1), 3 constants (1, 2, 100), and 4 labels (SORT, 200, 210, 220), so that $\eta_2 = 13$. Halstead further defined the metric *total occurrences of operands*, and named it N_2 (Equation (4)). It is a measure of the total amount of data used by the program. Table 1 shows that $N_2 = 42$.

The metrics *VAR*, η_2, and N_2 seem to be robust: slight variations in the schemes used to compute them do not seem to affect inordinately other Software Science measures based upon them. A more thorough discussion can be found in [SHEN83] or [CONT86].

3.2 The Usage of Data within a Module

Consider the Pascal bubble sort program shown in Figure 3. The program BUBBLE inputs two related integer arrays (A and B) of the same size up to 100 elements each. It uses a bubble sort on the A-array, interchanges the B-array values to keep them with the accompanying A-array values, and outputs the results. Figure 3 contains a main program in lines 15–38 and a subprogram procedure SWAP in lines 7-13. We can compute several metrics on the main

program and the subprogram. For example, $VAR = 7$ for the main program and $VAR = 3$ for procedure SWAP. In order to characterize the intra-module data usage, we may use the metrics *live variables* and *variable spans*, which are described below.

3.2.1 Live Variables

While constructing program BUBBLE, the programmer created a variable called LAST. Analyze Figure 3 carefully to see that all array elements beyond the "last" one are ordered. While the program is running, if SIZE = 25 and LAST = 14, then all items A[15] – A[25] and B[15] – B[25] are in order even though the first 14 elements of each array may not yet be ordered. A beginning value for LAST is established in line 20, decremented in line 24, and used in the logical expression in line 26.

There are only three statements in this program in which LAST appears, excluding the declaration in line 3. Does this mean that we do not need to be concerned with LAST while constructing the statements other than 20, 24, and 26? Certainly not. Between statements 20 and 26, it is important to keep in mind what LAST is doing. For example, statements 21–22 are used to set up a potentially never-ending loop. However, even though these statements never mention LAST, the programmer realized that each time an A-value bubbles down to its appropriate position, LAST will be decremented by one. Eventually on some cycle through the A-values none will be swapped and the loop beginning in statement 22 will be exited. As we will show later, LAST has a lifespan that begins at statement 20 and extends through statement 26.

Thus, a programmer must constantly be aware of the status of a number of data items during the programming process. A reasonable hypothesis is that the more data items that a programmer must keep track of when constructing a statement, the more difficult it is to construct. Thus, our interest lies in the size of the set of those data items called *live variables* (*LV*) for each statement in the program [DUNS79]:

A variable is called live from its first to its last reference within a procedure.

It is also possible to define the *average* number of live variables ($\overline{LV}$), which is the sum of the count of live variables divided by the count of executable statements in a procedure. This is an average measure for data usage in a procedure or program. The live variables in the program in Figure 3 appear in Table 2. The average live variables for this program is 110/28 = 3.9. The average live variables for the program in Figure 1 is 22/11 = 2.

As defined, live variables depend on the order of statements in the source program, rather than the dynamic execution-time order in which they are encountered. A metric based on run-time order would be more precisely related to the life of the variable, but would be much more difficult to define algorithmically (especially in a non-structured programming language).

3.2.2 Variable Spans

Two variables can be *alive* for the same number of statements, but their use in a program can be markedly different. For example, Figure 4 lists all of the statements in a Pascal program that refer to the variables A and B. Both variables are alive for the same 40 statements (21-60), but A is referred to three times while B is mentioned only once. A metric that captures some of the essence of how often a variable is used in a program is called the *span* (*SP*). This metric is the number of statements between two successive references to the same variable [ELSH76]. For a program that references a variable in n statements, there are $n-1$ spans for that variable. Thus, in Figure 4, A has 4 spans and B has only 2. Intuitively this tells us that A is being used more than B.

Furthermore, the size of a span indicates the number of statements that pass between successive uses of a variable. A large span can require the programmer to remember during the construction process a variable that was last used far back in the program. In Figure 4, A has 4 spans of 10, 12, 7, and 6 statements, while B has 2 spans of 23 and 14 statements. It is simple

to extend this metric to "average span size," in which case A has an average span size of 8.75 and B has an average span size of 18.5. It can be shown that the number of spans at a particular statement is also the number of live variables at that point.

4. LOGIC STRUCTURE METRICS

The logic structure of a program allows it to perform different operations dependent upon different input data or intermediate calculations. This differential processing is generally accomplished via the well-known and time-honored IF statement. For example, in Figure 1, depending on the values of X(I) and X(J), either the statements 10, 11 and 12 will be executed or they will be skipped. Thus, there is more than a single path through this program.

The structure in a program is often represented by a directed graph called a flowchart or flowgraph. It is conventional to highlight the points in a flowchart at which a test is performed and from which there are two or more possible branches. On the other hand, the actions taken in a branch after a test are often grouped together. Figure 5 is a flowchart for the program in Figure 1. The part of the algorithm dealing with the tests made and branches after tests is the "logic structure" of the program.

In a study in which experts rated complexity metrics, eight of nine product metrics considered important were related to measures of a program's logic structure [ZOLN81]. (The ninth important metric was related to data). Just like the metrics for data, there is no agreement on which logic metric is the most important. Even for metrics with the same name and intention, different researchers have used different counting schemes. The following sub-sections give the most popular counting methods for several representative metrics for the logic structure of programs.

4.1 Decision Count

The flow of control in a computer program normally proceeds sequentially. It is interrupted when statements such as IF, DO, WHILE, CASE and other conditional and loop statements are encountered. A very simple control structure metric *decision count, DE,* for a program is simply the count of these statements. Thus from Figure 1 *DE*=4. A program with a relatively larger *DE* is believed to be generally more complex than another program with a smaller *DE.*

Many programming languages allow the use of compound conditions. They can normally be converted to an equivalent sequence of simple conditions. For example, let c_1 and c_2 be two conditions and let s be a statement (or group of statements). The statement

IF c_1 AND c_2 THEN s

is equivalent to either

IF c_1 THEN IF c_2 THEN s

or

IF c_2 THEN IF c_1 THEN s

Thus, it is reasonable to count an IF statement with two simple conditions as contributing two to the count of decisions. These simple conditions, called *predicates,* should be counted as *DE* in a program instead of the number of occurrences of the keyword (in most languages the keyword IF) for conditional statements. If a tool to count the basic software metrics (such as the Software Science operators and operands) is available, it can easily be extended to include the count of decisions (*DE*) [SHEN85].

An apparently more sophisticated and better known metric based on the number of decisions is the *cyclomatic complexity number* ($v(G)$) proposed by McCabe [MCCA76]. This metric was originally designed to measure the number of "linearly independent" paths through

a program, which in turn was believed to relate to the testability and maintainability of the program. Since a program with a backward branch potentially has an infinite number of paths, a measure which is a count of the number of some distinct "basic" paths, rather than all possible paths, is probably more meaningful.

The cyclomatic complexity as defined by McCabe for a single program is

$$v(G) = e-n+2 \tag{6}$$

where e is the number of edges and n is the number of nodes. For example, consider the flowgraph in Figure 5. The rectangles are statements and diamond boxes are decision points in the program. The flowgraph has 12 nodes and 15 edges, leading to $v(G) = 15-12+2 = 5$. Note that the node count includes both rectangles and diamonds.

Although the construction and analysis of flowcharts from source code is not trivial, it can be shown [CONT86] that a relationship between $v(G)$ and DE exists which is simply

$$v(G) = DE+1 \tag{7}$$

A program with several modules has a flowgraph with several connected components. It can be shown that the cyclomatic complexity for a multi-module program is simply the sum of the $v(G)$'s for the individual modules. That is, for a program with m modules,

$$v(G_{\text{program}}) = \sum_{i=1}^{m} v(G_i) \tag{8}$$

where $v(G_i)$ is the cyclomatic complexity of the i^{th} module. Since $v(G_i) = DE_i + 1$,

$$v(G_{\text{program}}) = \sum_{i=1}^{m} DE_i + m. \tag{9}$$

4.2 Minimum Number of Paths and Reachability Metrics

Schneidewind and Hoffmann [SCHN79] defined metrics for the "minimum number of paths" (N_p) in a program and the "reachability" (R) of any node. In order to determine N_p, one approach is to describe each path – a unique sequence of arcs from the start node to the

terminal node. This path analysis also leads to the determination of R, which is the number of unique ways of reaching each node.

Obviously, the number of paths could be large (or infinite) when loops exist. The determination of N_p excludes paths with loops traversed more than once. As an example, consider again Figure 5. There are 5 unique paths (i.e., $N_p = 5$):

1,4,14
1,4,5a,5b,5c,14
1,4,5a,5b,5c,6,7a,7b,7c,5b,5c,14
1,4,5a,5b,5c,6,7a,7b,7c,8,7b,7c,5b,5c,14
1,4,5a,5b,5c,6,7a,7b,7c,8,9-11,7b,7c,5b,5c,14

This technique may be awkward to use on large programs. Shooman proposes a method to estimate N_p based on algebraic considerations (see Chapter 4 of [SHOO83]). The reachability R of each node can be determined from the information in Table 3. For example, Table 3 shows that there are 4 unique paths to node 5b but only one unique path to node 5a. The average reachability $\bar{R}$ is computed from the total number of paths 26 divided by the number of nodes 12.

In an attempt to validate the usefulness of N_p and $\bar{R}$, Schneidewind and Hoffman gathered data on 64 errors found during debugging and testing actual software products. They found high correlations between N_p and $\bar{R}$ and some measures of program errors. That is, the correlation coefficients were .76 (number of errors found and N_p), .77 (number of errors found and $\bar{R}$), .90 (time to find errors and N_p), and .90 (time to find errors and $\bar{R}$).

4.3 Nesting levels

"Nesting" allows the programmer to avoid excessive compound conditionals in any one IF or WHILE statement by taking advantage of conditions in effect due to previous IF or WHILE statements. However, it has been found that excessive nesting can lead to circumstances where it is actually difficult for programmers to comprehend what must be true for a particular statement to be reached [DUNS79].

Thus, another important complexity metric is the "depth of nesting" [ZOLN81]. For example, a simple statement in the sequential part of a program, such as line 5 in Figure 1, may be executed only once. A similar statement, such as line 9, is executed $O(n^2)$ times (n is the size of the array and stored in variable N) since it is part of an inner loop. The higher the depth or *nesting level*, the more difficult it is to assess the entrance conditions for a certain statement. Such concerns lead to the definition of the metric "average nesting level" ($\overline{NL}$) [DUNS80]. In order to compute this metric, every executable statement in a program must be assigned a nesting level. A simple recursive procedure for doing this is described as follows:

(1) The first executable statement is assigned nesting level 1.

(2) If statement a is at level l and statement b follows sequentially the execution of statement a, then the nesting level of statement b is l also.

(3) If statement a is at level l and statement b is in the range of a loop or a conditional transfer governed by statement a, then the nesting level of statement b is $l+1$.

Notice that all executable statements in Figure 1 have been assigned a nesting level (shown in the column labelled "Level") via this procedure. In order to determine the average nesting level ($\overline{NL}$), sum all statement nesting levels and divide by the number of executable statements. For this program (with 11 executable statements) the sum is 34 and the average nesting level is 3.1.

4.4 Transfer usage

In a classic letter denouncing the use of the GOTO statement, Dijkstra suggested that a wise programmer should strive to narrow the conceptual gap between the static program and the dynamic process which it represents [DIJK68]. That is, a program is easier to understand if "successive" statements in the program text also correspond to "successive" actions in time. The use of direct transfers (or GOTO statements) can disrupt such correspondence. In this

spirit, a metric for the uncontrolled use of GOTO statements is the measure *knots* proposed by Woodward, Hennell and Hedley [WOOD79]. They observed that during program development or debugging, a Fortran programmer often laid out the listing and proceeded to draw lines with arrows on the margin indicating the flow (or the interruption thereof) of control. The sample program in Figure 1, reproduced in Figure 6, shows the possible lines. In this case there is no knot as the two GOTO statements do not cross each other. Note that, in addition to forward transfers, backward arrows are drawn at the end of each loop. Suppose for some reason we modify the program by reversing the direction of the test in line 8 and add a GOTO statement shown in Figure 7. This is an awkward way to sort correctly, but the purpose is to show that this is a case where there is one knot. The definition of the knots metric can be stated precisely: Assume that the lines in a program are numbered sequentially. Let an ordered pair of integers (a,b) indicate that there is a direct transfer from line a to line b. Given two pairs (a,b) and (c,d), there is a knot if one of the following two cases is true:

(1) $\min(a,b) < \min(c,d) < \max(a,b)$

and $\max(c,d) > \max(a,b)$

(2) $\min(a,b) < \max(c,d) < \max(a,b)$

and $\min(c,d) < \min(a,b)$

The ordered pairs for the program in Figure 7 are (4,15) , (8,10) and (9,13). The pair (4,15) does not form a knot with any other pair; (8,10) and (9,13) satisfy case (1). This example shows that an inappropriate use of the conditional statement may increase the structural complexity, which is reflected by the knot count.

For languages that allow multiple statements on one line, it is necessary to reformat the source listing so that all direct transfers are on individual lines before applying the above definition to obtain the number of knots. There are also examples where it is possible to rearrange certain statements in a program without changing the function of the program, but which

will change the knot count [WOOD79]. For these equivalent programs, the ones with the lower knot count are believed to be designed better.

5. EFFORT AND DEFECT METRICS

The metrics described in Sections 2, 3, and 4 are computed from the software product itself. People normally use them (or a combination of them) to infer properties of the software development process. The two most important properties of the development process are probably the development effort and the defects remaining in the product after development. It is therefore important to define metrics for them.

5.1 Measuring Effort

The units and method of measurement of effort and cost are often divided into two categories: the *micro-level* of measurement for effort expended by individual programmers on small projects, and the *macro-level* of measurement for effort expended by teams of programmers on large projects.

Micro-level projects are defined as small projects usually completed by a single programmer in a few days or weeks. The most appropriate effort metrics are units of time in minutes, hours, or at most days. Since only one individual is involved, the time is directly convertible to the normal effort measure, the person-hour. Note that such projects are uncommon in industry, but are quite common in small organizations and very common in experiments that investigate various questions concerning software development.

For micro-level projects, the time spent on the design, coding and testing of the software product can be measured using an ordinary clock or stopwatch. Records should be made *during* the development process, rather than afterwards to avoid the introduction of errors. Such errors may be due to faulty recollection, or intentional bias toward the time the process *should* have taken. The times for the coding and testing phases can be substantiated with computer logs if a

terminal is used during software development. It is important to exclude from the elapsed time interruptions such as phone calls or coffee breaks.

While not very common, there are effort metrics other than time. For example, a metric such as the "number of runs submitted during development" could be treated as an effort metric. But, since it may be unclear how to relate such a metric to time and cost, and since it is usually just about as easy to collect time, time is typically the effort metric of choice.

Macro-level projects are defined as large projects usually completed by a team of at least two programmers in weeks, months or years. The most appropriate effort metrics are person-months or person-years. The measurement normally refers to the resources expended by professionals in the design, coding, and testing of the software product, including direct management and documentation activities. In industry this information can be collected directly during the software development process or can be recovered from the accounting records after the project is completed. Since each organization establishes its rules according to its own needs, consistency from organization to organization is uncommon. Thus, one should use caution in comparing programming effort for projects developed at different organizations, or at different times even within the same organization.

Macro-level effort differs from micro-level effort in another aspect. Although a programmer may be able to construct hundreds of lines of code per day for a small project, one cannot expect the intensity to persist over the months required to complete a team project. Furthermore, in a team situation, there is a significant number of necessary interruptions over a period of time. Times for coffee breaks, lunch, meetings with supervisors, meetings with other programmers, informal discussions with other programmers, sick days, vacations and personal business are normally charged to the project. One manager has observed that "typically only 4 or 5 hours of an eight-hour day are applied to the project" [PUTN78]. In addition, all overhead related to understanding specifications, techniques, and programming language features, as well

as unit testing are generally included. Thus, rates of programming productivity in small controlled experiments can seldom be applied to large team projects.

5.2 Measuring Defects

Defect metrics can pertain to the design, coding, testing and maintenance phases of the software life cycle. Design defects can be found during design reviews. Pre-release defects can be found during code review, unit test, system test activities. Post-release defects are found by the customers.

There are three general methods to measure the number of defects:

(1) Number of defect reports

(2) Number of program (or design) changes

(3) Number of changed lines of code

The first method may be misleading because the same defect may cause several failures. It is often difficult to go through the defect reports and algorithmically distinguish the independent ones and count them. The second method may be done algorithmically if we define a *program change* as

1. One or more changes to a single statement.
2. One or more statements inserted between existing statements.
3. A change to a single statement followed by the insertion of new statements.

However, it may not be "fair" to consider a change to insert a block of code to be *equivalent* to a change to fix a misspelling in an error message. The third method overcomes the problem by "weighting" the defects by the number of lines that are needed to fix them.

The number of defects for a software product is often used as a "quality" measure for the product. Since larger products are expected to have a higher number of defects, it is customary

to normalize the raw count of defects by the size of the software. That is, the most common metric for quality comparison purposes is

$$\text{defect density} = \text{number of defects}/S \tag{10}$$

where S is the program size in thousands of lines of code. However, the defect density seems to vary as a function of program size. It is shown in [CONT86] that this measure is actually inappropriate in many situations.

6. SOME APPLICATIONS OF SOFTWARE METRICS

6.1 Early Software Size Estimation

Software metrics are interesting for their own merit, but they take on a special significance and value when incorporated into a model and used as a predictive tool. In all of the major productivity models program size is not only required, but is the predominant parameter. Without a reliable size estimate the effort estimate cannot be accurate even if all of the other parameters are estimated accurately. However, empirical results (as well as anecdotal evidence) suggest that size estimation is nearly as difficult a procedure as effort estimation for programmers, system analysts, and managers [BOEH81, DEMA81].

Various techniques are employed for size estimation, but the predominant ones involve modular size estimating. The designer (based on experience and intuition) assigns an approximate size to each module and accumulates them to obtain an overall size estimate. The estimation process can be undertaken by a single "expert" manager or team leader, or it can be a group process in which one or more individuals estimate the size of each module and then combine estimates using some weighting system. Even so, the process is still predicated on the belief that people can estimate size more accurately than effort.

The problem may be solved in part if people can estimate the number of unique variables in a program more accurately than its size. This premise is plausible since several modern

development methodologies (and programming languages) require the programmer to design the data structure as the first step in the development. In [DUNS85] the authors discuss an experiment involving 44 subjects each constructing two Pascal programs on the order of a few hundred lines of code. In early milestone interviews for each program, subjects estimated the size of the program to be constructed (S_s) and the number of unique variables it would contain (*VAR*). From independent data they had developed a formula using *VAR* for predicting program size. The study showed a strong linear correlation between *VAR* and S_s. Similarly [ALBR83] reported a correlation between function points and S_s. Since both *VAR* and number of function points can be estimated early, this could lead to a useful technique for early size estimation.

6.2 Guiding the Testing Process

Models that estimate the number of defects share a common problem: the data collected to validate the models may not be precise and accurate. Therefore, we do not expect any model to estimate the number of defects with great accuracy. On the other hand, models that effectively discriminate between modules with defects and those without may be of great value. Such a model can be used to direct more testing resources to the modules that are determined to have defects, with the hope that the defects can be found and corrected before product release.

Potier *et al.* considered the basic Software Science metrics, the cyclomatic complexity, and the paths and reachability metrics in identifying the error-prone modules for a family of compilers [POTI82]. The approach was to compute the mean values of the complexity metrics for the set of procedures that had errors and for the set of procedures that did not. The "discriminant effect" of a metric was defined as the ratio of these mean metric values. For example, the 606 defect-free procedures and mean value $\overline{\eta} = 27.43$, while the remaining 500 modules with defects had $\overline{\eta} = 69.69$. A number between these two mean values may be used to decide whether a procedure will have defects. For example, using $\eta = 39$ as the threshold value, 485 out of 618 procedures with $\eta < 39$ were defect-free, and 367 out of 489 procedures

with $\eta > 39$ had defects. A technique called "non-parametric discriminant analysis" can be used to select a set of metrics and their threshold values to produce a decision tree.

Potier *et al.* found that it was very difficult to find a series of threshold values that correctly discriminate error-free and error-prone procedures. Furthermore, the threshold values probably will change from one application to another. It is encouraging, however, to see that the metric most effective for discrimination at the first level of the decision tree is $\eta(=\eta_1+\eta_2)$, which has the potential for being available *early* in the development process.

Shen *et al.* studied software programs developed at IBM's Santa Teresa Laboratory and released during the period between 1980 and 1983 [SHEN85]. They discovered that the metrics η_2 and *DE* were the best single predictors of the total number of defects and the probability that a module will have post-release defects. Since η_2 may be estimated early in the development process, it may be used to target certain modules for early or additional testing in order to increase the efficiency of the defect removal process.

6.3 Using Many Metrics to Determine Software Complexity

In [KAFU85] the authors attempt to determine if any of ten individual metrics or a combination of them provide information concerning errors and coding time of components of a software system. The metrics studied included LOC, Halstead's effort metric, McCabe's $v(G)$, an unweighted information flow complexity metric, an invocation complexity metric, a review complexity metric, a stability measure, and weighted information flow, review complexity and stability measures. In general, their findings indicate that there is a reasonable correlation between software metrics and errors and that in combinations the metrics can be applied in a consistent manner to gain insight into the software development process.

The central issue considered was whether or not error-prone components could be identified by software metrics used individually or in a group. The authors identified com-

ponents in their software database that had either a number of errors or a coding time at least two standard deviations above the average. Thirty-two such components were identified and these were called "outliers". It was then determined whether metric outliers were good indicators of error/coding time outliers. They found that of the ten metrics employed:

(a) 28 of 32 error outliers were identified by at least one metric,

(b) No one metric identified more than 20 of the outliers,

(c) 23 of the error outliers were correctly identified simultaneously by 3 or more metrics.

Furthermore, the study indicated that code metrics and structure metrics lead to different outliers which implies that code and structure metrics are measuring different properties of software components.

7. SUMMARY AND CONCLUSIONS

Sections 2 through 4 discussed metrics that are product-related. All of them can be derived from analyzing the software itself using an automatic tool. The most common metric is one that is related to program size – S_s (LOC), S (KLOC), or N (number of tokens). These metrics can be evaluated easily and objectively, except when parts of the program are adapted (or copied) from existing programs. There is no consensus on how adapted code should be counted. If a tool that works like the lexical phase of a compiler is available, then it is easy to make more detailed measurements. Metrics such as η_1, η_2, N_1, N_2, VAR, and DE can be evaluated in one pass. Again, the counts may be misleading if parts of the code are adapted.

Section 5 discussed effort and defect metrics. The number of code changes are frequently counted as program defects. The application of the primitive software metrics to early size estimation and their use in defect models were discussed in Section 6.

Research using software metrics as a basis is continuing. Systematic collection of these and other useful metrics is a necessary prerequisite if the software development process is ever

to achieve the status of an engineering discipline.

REFERENCES

[ALBR83] Albrecht, A. J., and J. E. Gaffney, Jr. Software function, source lines of code, and development effort prediction: a Software Science validation. *IEEE Transactions on Software Engineering* SE-9, 6 (November 1983), 639-647.

[BAIL81] Bailey, J. W. and V. R. Basili. A meta-model for software development resource expenditures. *Proceedings of the Fifth International Conference on Software Engineering* (1981), 107-116.

[BAKE72] Baker, F. T. Chief programmer team management of production programming. *IBM Systems Journal* 11, 1 (1972), 56-73.

[BASI79] Basili, V. R., and R. W. Reiter. An investigation of human factors in software development. *IEEE Computer* 12, 12 (December 1979), 21-38.

[BELL74] Bell, D. E. and J. E. Sullivan. Further investigation into the complexity of software. *MITRE Technical Report 2874-2* (June 1974).

[BOEH81] Boehm, B. W. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ (1981).

[CHRI81] Christensen, K., G. P. Fitsos, and C. P. Smith. A perspective on Software Science. *IBM Systems Journal* 20, 4 (1981), 372-387.

[CONT86] Conte, S. D., H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models*. Benjamin/Cummings, Menlo Park, CA (1986).

[DEMA81] DeMarco, T., Yourden project survey: final report. Yourden, Inc. New York, N.Y. (September 1981).

[DIJK68] Dijkstra, E. W. Go to statements considered harmful. *Communications of the ACM* 11, 3 (March 1968), 147-148.

[DUNS79] Dunsmore, H. E., and J. D. Gannon. Data referencing: an empirical investigation. *IEEE Computer* 12, 12 (December 1979), 50-59.

[DUNS80] Dunsmore, H. E. and J. D. Gannon. Analysis of the effects of programming factors on programming effort. *The Journal of Systems and Software* 1, 2 (1980), 141-153.

[DUNS85] Dunsmore, H. E. and A. S. Wang, A step toward early software size estimation for use in productivity models. Proceedings of the 1985 National Conference on Software Quality and Productivity, Williamsburg, Virginia (March 1985).

[ELSH76] Elshoff, J. L. An analysis of some commercial PL/1 programs. *IEEE Transactions on Software Engineering* SE-2, 2 (June 1976), 113-120.

[HALS77] Halstead, M. H. *Elements of Software Science*. Elsevier North-Holland, New York, NY (1977).

[KAFU85] Kafura D. and J. Canning, A validation of software metrics using many metrics and many resources. Department of Computer Science Report, Virginia Polytechnic Institute (1985).

[KNUT73] Knuth, D. E. *The Art of Computer Programming, Vol. 3: Searching and Sorting*. Addison Wesley, Reading, MA (1973).

[MCCA76] McCabe, T. J. A complexity measure. *IEEE Transactions on Software Engineering* SE-2, 4 (December 1976), 308-320.

[PAUL83] Paulsen, L. R., G. P. Fitsos, and V. Y. Shen. A metric for the identification of error-prone software modules. *TR 03.228,* IBM Santa Teresa Laboratory, San Jose, CA (June 1983).

[POTI82] Potier, D., J. L. Albin, R. Ferreol, and A. Bilodeau. Experiments with computer software complexity and reliability. *Proceedings of the Sixth International Conference on Software Engineering* (1982), 94-103.

[PUTN78] Putnam, L. H., A general empirical solution to the macro software sizing and estimating problem. *IEEE Transactions on Software Engineering* SE-4, 4 (July 1978): 345-361.

[SCHN79] Schneidewind, N. F., and H. Hoffmann. An experiment in software error data collection and analysis. *IEEE Transactions on Software Engineering* SE-5, 3 (May 1979), 276-286.

[SHEN83] Shen, V. Y., S. D. Conte, and H. E. Dunsmore. Software Science revisited: a critical analysis of the theory and its empirical support. *IEEE Transactions on Software Engineering* SE-9, 2 (March 1983), 155-165.

[SHEN85] Shen, V. Y., T. J. Yu, S. M. Thebaut, and L. R. Paulsen. Identifying error-prone software – an empirical study. *IEEE Transactions on Software Engineering* SE-11, 4 (April 1985), 317-324.

[SHOO83] Shooman, M. L. *Software Engineering.* McGraw-Hill, New York, NY (1983).

[SMIT80] Smith, C. P. A Software Science analysis of programming size. *Proceedings of the ACM National Computer Conference* (October 1980), 179-185.

[THEB83] Thebaut, S. M. *The Saturation Effect in Large-Scale Software Development: its Impact and Control.* Ph.D. Thesis, Department of Computer Science, Purdue University, West Lafayette, IN (May 1983).

[WOOD79] Woodward, M. R., M. A. Hennell, and D. Hedley. A measure of control flow complexity in program text. *IEEE Transactions on Software Engineering* SE-5, 1 (January 1979), 45-50.

[ZOLN81] Zolnowski, J. C. and D. B. Simmons. Taking the measure of program complexity. *Proceedings of the National Computer Conference* (1981), 329-336.

[WILS72] Wilson, R. J. *Introduction to Graph Theory.* Academic Press, New York, NY (1972).

[WOOD79] Woodward, M. R., M. A. Hennell, and D. Hedley. A measure of control flow complexity in program text. *IEEE Transactions on Software Engineering* SE-5, 1 (January 1979), 45-50.

[ZOLN81] Zolnowski, J. C. and D. B. Simmons. Taking the measure of program complexity. *Proceedings of the National Computer Conference* (1981), 329-336.

Figure 1. A Fortran subroutine that sorts an array into ascending order.

```
Line                                                                  Level
  1         SUBROUTINE SORT (X,N)
  2         INTEGER X(100),N,I,J,SAVE,IM1
  3       C THIS ROUTINE SORTS ARRAY X INTO ASCENDING
  4       C ORDER.
  5         IF(N.LT.2) GO TO 200                                          1
  6            DO 210 I=2,N                                               2
  7               IM1=I-1                                                 3
  8               DO 220 J=1,IM1                                          3
  9                  IF(X(I).GE.X(J)) GO TO 220                           4
 10                     SAVE=X(I)                                         5
 11                     X(I)=X(J)                                         5
 12                     X(J)=SAVE                                         5
 13    220           CONTINUE                                            3
 14    210         CONTINUE                                              2
 15    200      RETURN                                                   1
 16         END
```

Figure 2. A cross-reference listing of subroutine "sort".

X	1	2	9	9	10	11	11	12
N	1	2	5	6				
I	2	6	7	9	10	11		
J	2	8	9	11	12			
SAVE	2	10	12					
IM1	2	7	8					

Figure 3. A Pascal bubble sort program.

```
 1   program BUBBLE (input,output);
 2   type INTARRAY = array [1..100] of integer;
 3   var I, J, LAST, SIZE: integer;
 4       CONTINUE: boolean;
 5       A, B: INTARRAY;
 6
 7      procedure SWAP (var X: INTARRAY; K:integer);
 8      var  T: integer;
 9      begin
10         T := X[K];
11         X[K] := X[K+1];
12         X[K+1] := T
13      end;
14
15   begin
16      read (SIZE);
17      for J := 1 to SIZE do begin
18         read (A[J],B[J])
19      end;
20      LAST := SIZE;
21      CONTINUE := true;
22      while CONTINUE do begin
23         CONTINUE := false;
24         LAST := LAST -1;
25         I := 1;
26         while I <= LAST do begin
27            if A[I] > A[I+1] then begin
28               CONTINUE := true;
29               SWAP (A,I);
30               SWAP (B,I)
31            end;
32            I := I+1
33         end
34      end;
35      for J := 1 to SIZE do begin
36         writeln (A[J],B[J])
37      end
38   end.
```

Figure 4. Statements in a Pascal program referring to variables A and B.

```
          ...
21   read (A,B);
          ...
32   X := A;
          ...
45   Y := A-B;
          ...
53   Z := A;
          ...
60   writeln (A,B)
          ...
```

Figure 5. The control-flow graph for the program in Figure 1.

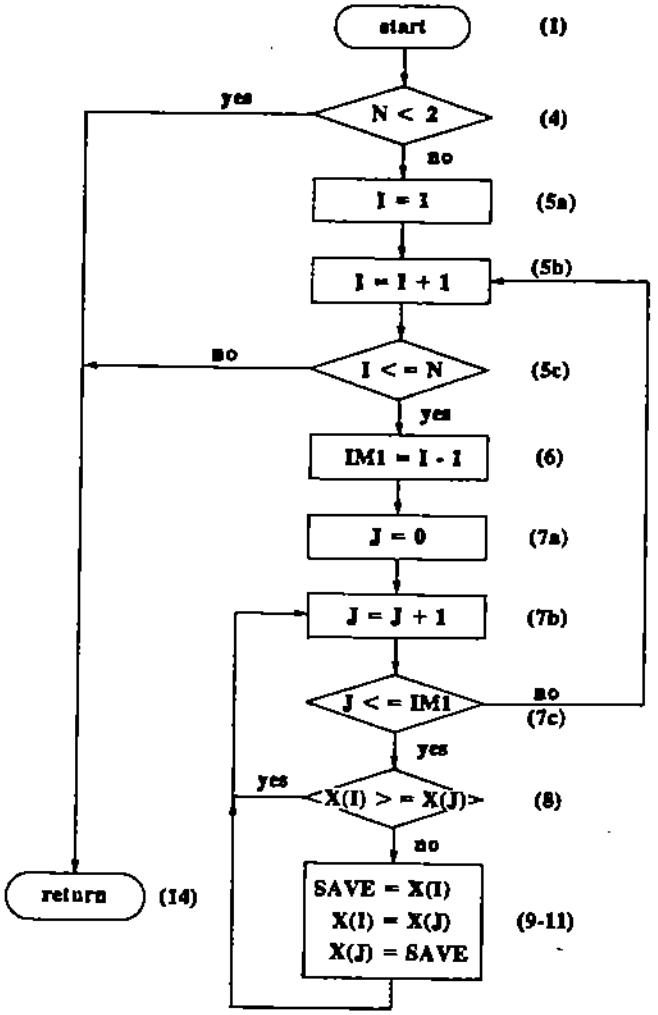
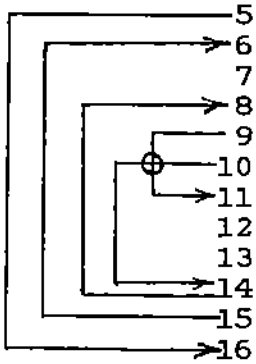


Figure 6. A Fortran subroutine that sorts an array into ascending order.

```
 Line                                                                  Level
   1          SUBROUTINE SORT (X,N)
   2          INTEGER X(100),N,I,J,SAVE,IM1
   3        C THIS ROUTINE SORTS ARRAY X INTO ASCENDING
   4        C ORDER.
   5          IF(N.LT.2) GO TO 200                                       1
   6             DO 210 I=2,N                                            2
   7                IM1=I-1                                              3
   8                DO 220 J=1,IM1                                       3
   9                   IF(X(I).GE.X(J)) GO TO 220                        4
  10                      SAVE=X(I)                                      5
  11                      X(I)=X(J)                                      5
  12                      X(J)=SAVE                                      5
  13   220             CONTINUE                                          3
  14   210          CONTINUE                                             2
  15   200       RETURN                                                  1
  16             END
```

Figure 7. A Fortran subroutine that sorts an array into ascending order.



```
Line                                                                    Level
  1        SUBROUTINE SORT (X,N)
  2        INTEGER X(100),N,I,J,SAVE,IM1
  3      C THIS ROUTINE SORTS ARRAY X INTO ASCENDING
  4      C ORDER.
  5        IF(N.LT.2) GO TO 200                                          1
  6           DO 210 I=2,N                                               2
  7              IM1=I-1                                                 3
  8              DO 220 J=1,IM1                                          3
  9                 IF(X(I).LT.X(J)) GO TO 230                           4
 10                    GO TO 220
 11    230             SAVE=X(I)                                        5
 12                    X(I)=X(J)                                        5
 13                    X(J)=SAVE                                        5
 14    220          CONTINUE                                            3
 15    210       CONTINUE                                               2
 16    200    RETURN                                                    1
 17           END
```

Table 1. A token analysis of subroutine SORT.

Operators	Occurrences	Operands	Occurrences
SUBROUTINE	1	X	8
()	10	N	4
,	8	I	6
INTEGER	1	J	5
IF	2	SAVE	3
.LT.	1	IM1	3
GO TO	2	1	2
DO	2	2	2
=	6	100	1
-	1	SORT	1
.GE.	1	200	2
CONTINUE	2	210	2
RETURN	1	220	3
end-of-line	13		
$\eta_1=14$	$N_1=51$	$\eta_2=13$	$N_2=42$

Table 2. Live Variables for the Program in Figure 3.

Line	Live Variables	Count
9		0
10	T, X, K	3
11	T, X, K	3
12	T, X, K	3
13		0
15		0
16	SIZE	1
17	SIZE, J	2
18	SIZE, J, A, B	4
19	SIZE, J, A, B	4
20	SIZE, J, A, B, LAST	5
21	SIZE, J, A, B, LAST, CONTINUE	6
22	SIZE, J, A, B, LAST, CONTINUE	6
23	SIZE, J, A, B, LAST, CONTINUE	6
24	SIZE, J, A, B, LAST, CONTINUE	6
25	SIZE, J, A, B, LAST, CONTINUE, I	7
26	SIZE, J, A, B, LAST, CONTINUE, I	7
27	SIZE, J, A, B, CONTINUE, I	6
28	SIZE, J, A, B, CONTINUE, I	6
29	SIZE, J, A, B, I	5
30	SIZE, J, A, B, I	5
31	SIZE, J, A, B, I	5
32	SIZE, J, A, B, I	5
33	SIZE, J, A, B	4
34	SIZE, J, A, B	4
35	SIZE, J, A, B	4
36	J, A, B	3
37		0

Table 3. The Reachability (R) of each node in Figure 5.

node	R	paths
1	1	(1)
4	1	(1,4)
5a	1	(1,4,5a)
5b	4	(1,4,5a,5b) (1,4,5a,5b,5c,6,7b,7c,5b)
		(1,4,5a,5b,5c,6,7a,7b,7c,8,7b,7c,5b)
		(1,4,5a,5b,5c,6,7a,7b,7c,8,9-11,7b,7c,5b)
5c	4	(1,4,5a,5b,5c) (1,4,5a,5b,5c,6,7a,7b,7c,5b,5c)
		(1,4,5a,5b,5c,6,7a,7b,7c,8,7b,7c,5b,5c)
		(1,4,5a,5b,5c,6,7a,7b,7c,8,9-11,7b,7c,5b,5c)
6	1	(1,4,5a,5b,5c,6)
7a	1	(1,4,5a,5b,5c,6,7a)
7b	3	(1,4,5a,5b,5c,6,7a,7b) (1,4,5a,5b,5c,6,7a,7b,7c,8,7b)
		(1,4,5a,5b,5c,6,7a,7b,7c,8,9-11,7b)
7c	3	(1,4,5a,5b,5c,6,7a,7b,7c) (1,4,5a,5b,5c,6,7a,7b,7c,8,7b,7c)
		(1,4,5a,5b,5c,6,7a,7b,7c,8,9-11,7b,7c)
8	1	(1,4,5a,5b,5c,6,7a,7b,7c,8)
9-11	1	(1,4,5a,5b,5c,6,7a,7b,7c,8,9-11)
14	5	(1,4,14) (1,4,5a,5b,5c,14) (1,4,5a,5b,5c,6,7a,7b,7c,5b,5c,14)
		(1,4,5a,5b,5c,6,7a,7b,7c,8,7b,7c,5b,5c,14)
		(1,4,5a,5b,5c,6,7a,7b,7c,8,9-11,7b,7c,5b,5c,14)
$\overline{R}$ =26/12 =2.2 (average reachability)		